KU-ACM

Programming Competition

Languages supported:

C

C++

Go

Haskell

Java

JavaScript

Perl

PHP

Prolog

Python

Racket

Ruby

Number of Problems: 10

Saturday, April 6th

11:00AM to 5:00PM

(first hour for lunch/orientation)

Problem 1: **(filter web)**

As a programmer, you find unbalanced parentheses unsightly, so you decide to write a program to shield your eyes whilst browsing the interwebs. You figure if the author couldn't balance their parentheses, they probably didn't have anything good to say, so you might as well discard their message.

A balanced string is a string for which each and every '(' has a matching ')' somewhere to its right. We are programmers, so parentheses may be nested.

Input: write a program that reads multiple strings, separated by newlines, from stdin.

Output: If the parentheses in the string are balanced, or if it contains no parentheses, print 'balanced' to stdout; otherwise print out 'unbalanced'. Each entry should be separated by newlines.

sample input	***sample output***
`(word (with friends))`	`balanced`
`)()BALANCED?(()`	`unbalanced`
`ever and forever and for`	`balanced`
`o()xxx[{::::::::::::>`	`balanced`
`quoted ones still break "("`	`unbalanced`

Problem 2: **Range Table**

You're a naval engineer in a Napoleonic-era alt-history. In your time, mathematical and scientific advances are driven by the necessities of warfare, and advancements in programmable mechanical difference engines have allowed the rapid production of range tables for use by His Majesty's cannoneers. The Ministry of Applied Sciences has tasked you with creating such a table. Luckily, the staff of this competition can translate whichever programming language you choose into the correct sequence of punch cards.

Write a program that will calculate the angle necessary to fire a given distance. The equation you will be utilizing is as follows:

$$d = \frac{v^2 sin(2\theta)}{g}$$

where
- ***d*** *= distance traveled (non-negative)*
- ***v*** *= initial velocity (non-zero, positive)*
- ***g*** *= 9.8 (constant)*
- ***Θ*** *= angle of launch* ***in degrees***

Input: Read from stdin successive pairs of space-delimited integers, one pair per line, that represent the distance the object must travel in meters (***d***) and its (non-zero) initial velocity in meters/second (***v***). Each set of values will be separated by a newline.

Output: Send to stdout the necessary angle (***Θ***) for each line, one answer per line. Keep 4 decimals. If a distance is unreachable for a given starting velocity with any angle, print 'no possible angle' instead. ***Θ*** should be between 0 and 90—we do not want to fire backwards.

sample input	***sample output***
`100 35`	`26.5651`
`110 40`	`21.1786`
`200 45`	`37.7219`
`5000 1`	`no possible angle`

Problem 3: **InterLOLation**

Ever wonder what halfway between Happy Cat and President Obama is? Of course you have, time to make it happen. Write a program to take in two images of the same dimensions and output an image where each pixel is the average of the two corresponding pixels in the original image. The images will be of the following format:

All numbers are integers.
The first line gives the space-separated dimensions of the image in pixels, width then height. The second line contains RGB space-separated intensity information for each pixel, starting from upper-left, going across, continuing left-to-right on the next row, and so on until the final, bottom-right pixel. Minimum intensity is 0; maximum intensity is 255.

Example:

```
2 2
0 0 0 127 127 127 63 63 63 191 191 191
```

Note that each pixel is represented by three integers: <Red> <Green> <Blue>. These two lines represent the image (with scale significantly increased):

Input: read from stdin the raw text of two images that will be separated by a blank line.

Output to stdout the text of a merged image, formatted to the same specification, which has for its pixel values the average of each pixel of the two files.

***Note**: when dividing, perform floored division (e.g.* `255/2 = 127`*). File pairs will always be of the same dimensions.*

sample input

```
2 2
0 255 0 15 0 0 0 0 0 0 0 0

2 2
0 0 0 63 63 63 0 0 0 63 9 0
```

sample output

```
2 2
0 127 0 39 31 31 0 0 0 31 4 0
```

Problem 4: **> trash find**

Oh no, you've lost your terminal password in a collection of garbage text! Luckily, each individual character of the password was written in hexadecimal form, which ought to make it easier to find in a collection of other characters. Write a program that:

1. reads a line of text from stdin,
2. searches for valid hexadecimal characters,
3. converts those values to their ASCII counterparts,
4. concatenates the ASCII characters together to recover the password, then
5. outputs it to stdout.

For example, if the password was '42' then it might be hidden in a pile of text like this:

`jsdiDSnweoinx340923knasounqxp0djsfweoinax32jafowe09jvq209rjwefoji`

where **x34** and **x32** represent the characters '4' and '2'.

Input: Hexadecimal characters in this particular format consist of an 'x' (lowercase only) followed directly by two hexadecimal digits [0-9a-fA-F]. Ensure you do not read non-hexadecimal digits: sequences of characters preceded by an x but followed by a letter not included in the hexadecimal digit set (any letter beyond f/F). The password will always consist of printable, non-whitespace ASCII characters (x21-x7E).

Output: Send the converted password to stdout.

sample input 1	***sample output 1***
`ax43esw2xjx23x`	`C#`

sample input 2	***sample output 2***
`67x67q67 x6FXF6`	`go`

sample input 3	***sample output 3***
`x43tx4Frx42ox4flox4clo`	`COBOL`

Problem 5: **Geocache**

Geocaching is a recreational treasure hunting activity involving the use of GPS technology and clues as a means of locating hidden items, or *caches*. Geocachers may post direct coordinates or choose to give the same information in a subtler manner. You're on the hunt, and the dropper of a particular geocache tells you that the prize is hidden in the "centroid" of a park. Upon investigation you find that the park is a square flat plane with large granite blocks (rectangular prisms) at different locations relative to small fountain in the center, which will serve as the origin of a Euclidean (x,y) coordinate system. The centroid on a particular axis for n blocks can be calculated with the formula:

$$C_c = \frac{1}{M}\sum_{i=1}^{n} r_i m_i$$

where

C_c is the centroid coordinate of masses 1-n on a particular axis,
M is the total combined mass of objects 1-n,
r_i is the distance from object *i*'s projection on the axis to the origin,
m_i is the mass of object *i*.

The granite blocks are too big to lift onto scales and weigh individually, but luckily, you know them blocks to be of uniform composition with a density of 2750 kg/m^3. Assume a block (and thus its mass) is *centered* at its associated coordinates.

Input: you will read from stdin the x and y coordinates (relative to the fountain/origin) and height, width, and depth of each of a set of cube-shaped blocks. Each integer value for the x coordinate, y coordinate, and the 3 block dimensions will be separated by a space. Each block will be separated by a newline.

Output: print to stdout the x and y coordinates of the calculated centroid with the coordinates separated by a space.

The cache is a large one: for the sake of simplicity, always use integer division to obtain the final coordinates. Your answer should be a pair of integers.

sample input

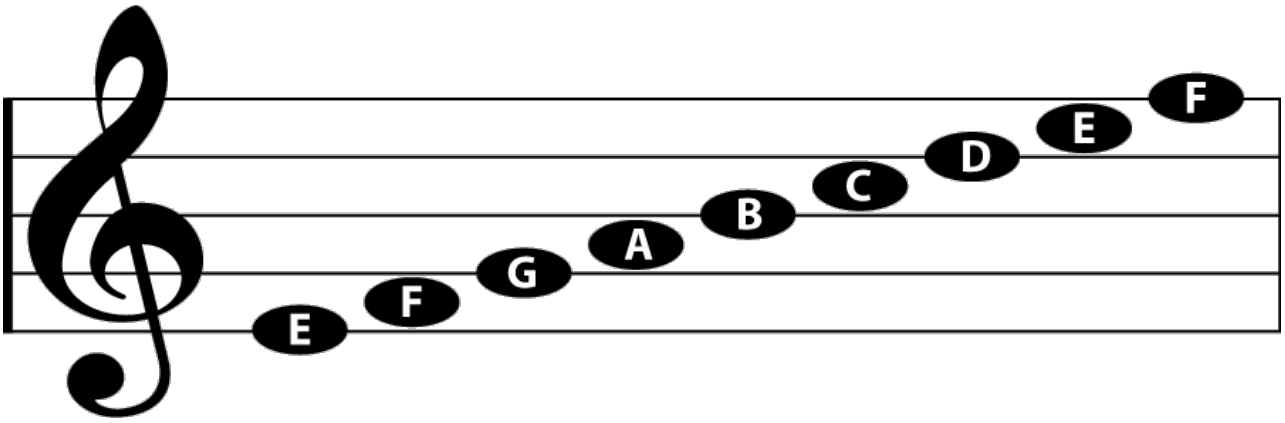
```
2 8 3 2 2
-2 -6 6 1 2
0 1 1 1 1
```

sample output

```
0 1
```

Problem 6: **Der Braillekönig**

Towards the end of Franz Schubert's life, he had gone both deaf and blind, so he converted all his sheet music to braille, using braille letters and symbols to stand for each note. Schubert wasn't a fan of documentation though, so he always omitted the key signatures from his pieces. Despite your modest knowledge of music, the Royal Museum has tasked you with determining the key of each piece of music. Music is normally arrayed on a staff like so:



Luckily, Ada Lovelace produced a braille reading routine that converts each piece of sheet music to a stream of characters, where E, F, G, A, B, C, D represent the notes.

Notes are encoded with further information using a prefix and suffix system. The beat prefix indicates how long the note is held. Whole notes are preceded by a 'w', half notes by an 'h', quarter notes by a 'q', eighth notes by an 'e', and sixteenth notes be an 's'. Rests are denoted by an 'r' and accept the same beat prefixes as notes. A note is 'sharp' if it is followed by a '#', and it is flat if followed by a lowercase 'b'. Thanks, Ada!

Each major key signature has the following characteristics: (Gist: http://bit.ly/13ZKpW1)

C major - (no sharps or flats)
G major - F#
D major - F#, C#
A major - F#, C#, G#
E major - F#, C#, G#, D#
B major - F#, C#, G#, D#, A#
F# major - F#, C#, G#, D#, A#, E#
C# major - F#, C#, G#, D#, A#, E#, B#

F major - Bb
Bb major - Bb, Eb
Eb major - Bb, Eb, Ab
Ab major - Bb, Eb, Ab, Db
Db major - Bb, Eb, Ab, Db, Gb
Gb major - Bb, Eb, Ab, Db, Gb, Cb
Cb major - Bb, Eb, Ab, Db, Gb, Cb, Fb

For instance, a stream of music that consists entirely of the characters:

```
qF#eF#qEeDqEeDeC#qBqAeG#eAeC#eEeA
```

would be in A major, since it contains F#, C#, and G#.

We know each piece of music to be complete, to contain all notes in its scale, and to be in a major key (Schubert was *insane*ly happy). For example, if by the end of a piece you have encountered a Bb and an Eb but no additional flats, you may assume the key to be Bb major.

Input: stdin will consist of successive streams of music, separated by newlines.

Output should be the key of the music as named in the above table (that is, without quotes), separated by newlines. You may assume the key never changes and that there are no double sharps/flats. Assume that all pieces of music correspond to the above grammar (i.e. there are no errors in the input).

sample input	***sample output***
`sCsDsEsFsGsAsBsC`	`C major`
`hBqA#qG#hF#hEhD#qC#qF#wB`	`B major`

Problem 7: **3n+1**

I. Definition of *next*:

for $x \in \mathbb{Z}^+$,

$$next(x) = \begin{cases} x/2 & \text{if } x \text{ is even, or} \\ 3x+1 & \text{if } x \text{ is odd.} \end{cases}$$

II. Characteristics of *next*:

Since $next(1) = 4$, $next(next(1)) = 2$, and $next(next(next(1))) = 1$, we may conclude that: $[next^*(x)] = [x, next(x), next(next(x)) = next^2(x), \ldots]$ is a bounded, divergent, infinite sequence for any x where recursive applications of $next$ eventually reach 1. We (the judges) have verified that this is the case for all $x \in$ [1..10,000,000].

III. Definition of *embiggen*:

Given two integers m and n, *where* m,n ≤ 10,000,000 *and* 0 ≤ n-m ≤ 10,000,

$$embiggen(m,n) = x$$

where m ≤ x ≤ n, such that the length of $[next^*(x)]$, up to and including the first occurrence of 1, is maximized. If more than one x maximizes the length of $[next^*(x)]$, choose the smallest x.

Example: $embiggen(1,8) = 7$, since:

$[next^*(1)]$ = [**1**, 4, 2, 1, …]
$[next^*(2)]$ = [**2, 1**, 4, 2, 1, …]
$[next^*(3)]$ = [**3, 10, 5, 16, 8, 4, 2, 1**, 4, 2, 1, …]
$[next^*(4)]$ = [**4, 2, 1**, 4, 2, 1, …]
$[next^*(5)]$ = [**5, 16, 8, 4, 2, 1**, 4, 2, 1, …]
$[next^*(6)]$ = [**6, 3, 10, 5, 16, 8, 4, 2, 1**, 4, 2, 1, …]
$[next^*(7)]$ = [**7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1**, 4, 2, 1, …]
$[next^*(8)]$ = [**8, 4, 2, 1**, 4, 2, 1, …]

IV. Desired program:

Input: stdin will consist of space-delimited pairs of integers m and n, one pair per line.

Output: print the value of $embiggen(m,n) = x$, one answer per line.

sample input	***sample output***
`1 8`	`7`
`20 40`	`27`
`1 100`	`97`

Problem 8: **Hackman**

You're a cyber hacker in the not-so-distant apocalyptic future. Unfortunately, you were discovered after your latest breach because you got distracted playing with your cat and didn't disconnect from the server you infiltrated. The police are en route, and you need to get to your ship as fast as possible. There are several teleporters on the way to your ship, but they might not get you closer. Plot the shortest route to your ship.

Input:
The first line has 2 numbers: ***h*** and ***w***, height and width of a grid, each in the range [5..500]
The next part of the input is h lines each with w characters. This grid represents the map, which is described by the following characters:

- 'S' : Your starting position (there will be only one S)
- '@': Your ship (there will be only one ship)
- ' ' (space) : Land
- '*': Walls
- (optional) numbers, '1', '2', … '9' that represent teleporters, with:
- (paired) lowercase letters 'a', 'b', … 'i' that represent the respective endpoints of teleportation (1 -> a, 2 -> b, …)

You can move one space at a time, in one of four directions (up, down, left, or right) so long as that space contains land, a teleporter, or your ship. You can assume there can only be up to 9 teleporters, mapping the numbers ***1***-***9*** to the lowercase letters ***a***-***i***. Moving onto a teleporter *instantly* moves you to the corresponding exit (e.g. moving onto a '1' transports you to the spot containing 'a'); teleporters do not work in reverse—in fact letters function as walls unless you have just teleported to them. You may not leave the grid (even if no wall/letter prevents you from doing so), and anyways moving off one edge would not take you to the opposite edge--this is Hackman, not Pacman.

Output:
Your moves as a single string composed of 'u' for up, 'd' for down, 'r' for right, 'l' for left.
i.e. ududrrll, where the moves are issued from left to right.

sample input 1

```
5 5
**S**
*   *
*** *
*@  *
*****
```

sample output 1

```
drddll
```

sample input 2

```
10 7
*s*****
* *b*a*
*     *
*1*** *
* * 2 *
* * * *
*     *
* *@* *
*  *  *
*******
```

sample output 2

```
ddddddddlld
```

sample input 3

```
13 15
*******s*******
*             *
***12345678** *
************* *
****hgfedcba* *
******* * * * *
*9* * *** * * *
*             *
* *********** *
*     *       *
* **  * ********
i@*           *
***************
```

sample output 3

```
dlldddd11111111ur
```

Problem 9: **Caesaris Sum**

Anglo-Julius Caesar lived in a time of danger and intrigue. In order to survive, he decided to create one of the first ciphers. You are a sub-captain of Caesar's army, and it is your job to decipher the messages sent by Caesar and provide them to your general. The original cipher was simple: for each letter in a plaintext message, you shifted it five places to the right to create the secure message (i.e., if the letter is 'A', the cipher text would be 'F', 'b' would be 'g', etc.).

Unfortunately, Caesar has been getting pretty paranoid, and has been sending messages shifted arbitrarily (anywhere from 0 to 25 rotations). Luckily, you have a scholar accompanying you who happens to have a table of letter frequencies in the English language. The proper decoding will be the one that minimizes the "distance" between observed and expected letter frequencies:

$$D = \sum_{\beta=\mathrm{a}}^{\mathrm{z}} \left| \%\beta_{expected} - \%\beta_{observed} \right|$$

Input/Output: write a program that takes in a text with shifted words and outputs the properly decoded original message. Any letter shifted past Z wraps around alphabetically (e.g. 'Z' shift 2 == 'B'), and upper/lower case should be preserved. *Assume spaces and any non [a-zA-Z] characters are not affected by the cipher*. Here is the table of letter frequencies; assume the message is long enough for the below table to provide reasonably high accuracy:

a	8.167%
b	1.492%
c	2.782%
d	4.253%
e	12.702%
f	2.228%
g	2.015%
h	6.094%
i	6.966%

j	0.153%
k	0.772%
l	4.025%
m	2.406%
n	6.749%
o	7.507%
p	1.929%
q	0.095%
r	5.987%

s	6.327%
t	9.056%
u	2.758%
v	0.978%
w	2.360%
x	0.150%
y	1.974%
z	0.074%

save typing:
http://bit.ly/14UFMMs

sample input

```
Mu xebt jxuiu jhkjxi je ru iubv-ulytudj, jxqj qbb cud qhu
shuqjut ugkqb, jxqj jxuo qhu udtemut ro jxuyh Shuqjeh myjx
suhjqyd kdqbyudqrbu Hywxji, jxqj qcedw jxuiu qhu Byvu, Byruhjo
qdt jxu fkhikyj ev Xqffyduii.--Jxqj je iuskhu jxuiu hywxji,
Weluhdcudji qhu ydijyjkjut qcedw Cud, tuhylydw jxuyh zkij femuhi
vhec jxu sediudj ev jxu weluhdut, qdt buj jxu weet jycui hebb,
$$$Mxqjsxq!21hbaz12h@~`
```

sample output

```
We hold these truths to be self-evident, that all men are
created equal, that they are endowed by their Creator with
certain unalienable Rights, that among these are Life, Liberty
and the pursuit of Happiness.--That to secure these rights,
Governments are instituted among Men, deriving their just powers
from the consent of the governed, and let the good times roll,
$$$Whatcha!21rlkj12r@~`
```

Problem 10: **Ad-MIN-istrator (get it?!)**

Imagine Johannes Gutenberg never invented the printing press, and scribes are still recording and copying data by hand. You oversee a corporation of scribes who make copies of books, and you are looking to increase your profits by distributing the work optimally. You have a single stack of n books, each with a varying number of pages, and m scribes. The scribes can all read and transcribe work at the same rate.

The book stack is pretty big, and you can't take books from anywhere but the top without disturbing the stack. You want to do book assignments for a particular scribe all at once, so you must choose some number of books from the top of the stack and assign them to that scribe. Luckily, you know exactly how many pages every book has.

Write a program that reads from stdin the following:
The first line contains the number of books, n, and the number of scribes, m.
This is followed by n lines with a number on each line representing the pages in each book.
If each page takes one unit of time to transcribe, output the minimum units of time it will take to transcribe all of the books if optimally distributed between the scribes. ***Large examples may overwhelm naïve algorithms; maximum runtime is 1 minute!***

sample input 1

```
4 5
14
15
16
17
```

sample output 1

```
17
```

sample input 2

```
7 3
12
25
30
5
1
18
19
```

sample output 2

```
37
```